	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorio de Computacion Salas A y B

Profesor(a): Rene Adrian Davila Perez

Asignatura: Programacion orientada a objetos

Grupo: 01

No de practica(s): practica 56

Integrante(s): 322001549, 322006829, 322125533, 425131152, 322103351

No de lista o brigada: 07

Semestre: 2026-1

Fecha de entrega: Viernes 3 de octubre de 2025

Observaciones:

Calificacion:

Índice

1. Introducción

La abstracción es la propiedad que considera los aspectos más significativos de un problema y expresa una solución en esos términos, mientras que encapsular, es cuando se aplica una çapa protectora.[a] un objeto de tal modo que, los atributos del mismo estén protegidos restringiendo el acceso, el resto del programa no puede acceder de forma directa a sus datos, por lo que es necesaria la aplicación de los métodos del objeto. Los métodos aseguran que se pueda acceder a los datos de forma adecuada.

Anteriormente se menciona el encapsulamiento y la abstracción que van directamente relacionadas con la jerarquía de clases. La API de Java contiene una inmensa variedad de paquetes, que a su vez contienen clases agrupadas para un mismo propósito, la organización de clases permitirá ayudar a las personas a encontrar una clase en particular. Debido a que la biblioteca de java contiene miles de clases, es necesaria alguna estructura para organizar la biblioteca facilitando el trabajo.

1.1. Planteamiento del problema

Se presenta un programa Java que simule un carrito de compras que cumplan con las siguientes características: añadir productos, eliminar productos mediante nombre o selección y limpiar el carrito, al cual, se implementa encapsulamiento y empaquetamiento con el *Full Qualified Name*.

1.2. Motivación

El uso del empaquetamiento facilita la organización del proyecto en módulos lógicos, lo que evita conflictos de nombres, mejora la legibilidad y fomenta la reutilización de componentes.

1.3. Objetivos

- Completar la implementación vista en clase.
- Inclusión del encapsulamiento, principalmente en la clase Articulo.
- La aplicación debe estar empaquetada con el *Full Qualified Name* ***mx.unam.fi.poo.p56.***

2. Marco teórico

2.1. Clase

Una clase en Java define un nuevo tipo de datos que puede incluir campos (variables) y métodos para definir el comportamiento de los objetos creados a partir de la clase: [1]

```
class ClassName {
    // Fields
    dataType fieldName;

    // Constructor
    ClassName(parameters) {
        // Initialize fields
    }

    // Methods
    returnType methodName(parameters) {
        // Method body
    }
}
```

De donde:

- ClassName: El nombre de la clase.
- fieldName: Variables que contienen el estado de la clase.
- Constructor: Método especial utilizado para inicializar objetos.
- methodName: Funciones definidas dentro de la clase para realizar acciones.

2.2. Objeto

Un objeto es una instancia de una clase. Se crea utilizando la palabra clave new seguida del constructor de la clase: [1]

```

ClassName objectName = new ClassName(arguments);
```

De donde:

- objectName: El nombre del objeto.
- arguments: Valores pasados al constructor para inicializar el objeto

A continuación un ejemplo incluyendo ambos conceptos:

```java

class Car {
    // Fields
    String color;
    int year;

    // Constructor
    Car(String color, int year) {
        this.color = color;
        this.year = year;
    }

    // Method
    void displayInfo() {
        System.out.println("Car color: " + color + ", Year: " +
    year);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object
        Car myCar = new Car("Red", 2020);
        myCar.displayInfo(); // Output: Car color: Red, Year: 2020
    }
}
```

En este ejemplo, la clase Car tiene los campos color y year, un constructor para inicializar estos campos, y un método displayInfo() para mostrar los detalles del coche. Se crea un objeto myCar utilizando la clase Car.

2.3. Constructor

En Java, un constructor es un método especial utilizado para inicializar objetos. A diferencia de los métodos normales, los constructores se invocan cuando se crea una instancia de una clase. Tienen el mismo nombre que la clase y no tienen tipo de retorno. Los constructores son esenciales para establecer los valores iniciales de los atributos del objeto y prepararlo para su uso.[2]

Tipos de constructores:

1. Constructor por defecto: El compilador Java proporciona automáticamente un constructor por defecto si no se definen constructores explícitamente en la clase. Inicializa el objeto con valores por defecto.
2. Constructor sin argumentos: Un constructor sin argumentos es definido explícitamente por el programador y no recibe ningún parámetro. Es similar al constructor por defecto, pero puede incluir código de inicialización personalizado.[2]
3. Constructor parametrizado: Un constructor parametrizado acepta argumentos para inicializar un objeto con valores específicos. Esto permite una inicialización más flexible y controlada de los atributos de los objetos.[2]
4. Copiar Constructor: Un constructor de copia se utiliza para crear un objeto nuevo como copia de un objeto existente. Java no proporciona un constructor de copia por defecto; sin embargo, puede implementarse manualmente.[2]

A continuación sus sintaxis de constructor por defecto:

```
class ClassName {
    // Constructor
    ClassName() {
        // Initialization code
    }

    // Parameterized Constructor
    ClassName(dataType parameter1, dataType parameter2) {
        // Initialization code using parameters
    }

    // Copy Constructor
    ClassName(ClassName obj) {
        // Initialization code to copy attributes
    }
}
```

Ejemplo de constructor:

```
public class Car {
    String model;
    int year;

    // Default Constructor
    public Car() {
        model = "Unknown";
        year = 0;
    }

    public static void main(String[] args) {
        Car car = new Car();
        System.out.println("Model: " + car.model + ", Year: " +
    car.year);
    }
}
```

En este ejemplo, la clase Car tiene un constructor por defecto que inicializa los atributos model y year con valores por defecto. Cuando se crea un objeto Car, se le asignan estos valores por defecto. [2]

Ejemplo de constructor parametrizado:

```
public class Car {
    String model;
    int year;

    // Parameterized Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public static void main(String[] args) {
        Car car = new Car("Toyota", 2021);
        System.out.println("Model: " + car.model + ", Year: " +
    car.year);
    }
}
```

Ejemplo de sobrecarga de constructores:

```java
public class Car {
    String model;
    int year;

    // No-Argument Constructor
    public Car() {
        model = "Unknown";
        year = 0;
    }

    // Parameterized Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car("Honda", 2022);
        System.out.println("Car1 -> Model: " + car1.model + ",
    Year: " + car1.year);
        System.out.println("Car2 -> Model: " + car2.model + ",
    Year: " + car2.year);
    }
}
```

Este ejemplo demuestra la sobrecarga de constructores en la clase Car, donde se definen tanto un constructor sin argumentos como un constructor parametrizado. Esto permite crear objetos con valores predeterminados o específicos.[2]

2.4. Métodos, parámetros y argumentos

2.4.1. Métodos

Los métodos son bloques de código que se utilizan para realizar tareas específicas. Cada método tiene un nombre que lo identifica y puede recibir una serie de parámetros, los cuales son variables que proporcionan información necesaria para que el método realice su tarea. Estos parámetros actúan como entrada para el método, permitiéndole operar con los valores recibidos.[3]

Dentro del cuerpo del método, se pueden realizar diversas operaciones, como cálculos matemáticos, manipulación de datos o incluso llamadas a otros métodos. Una vez que el método ha completado sus operaciones, puede devolver un resultado utilizando la palabra clave return seguida del valor que se desea devolver.[3]

La capacidad de reutilizar código es una de las ventajas clave de los métodos en Java. Al definir un método una vez, podemos llamarlo en diferentes partes de nuestro programa, evitando la repetición innecesaria de código. Esto no solo simplifica nuestro código, sino que también facilita su mantenimiento y permite una mayor modularidad en nuestra aplicación.[3]

```java
public tipo_de_dato_devuelto nombre_metodo(tipo_de_dato parametro1
    , tipo_de_dato parametro2) {
    // codigo a ejecutar
    return resultado;
}
```

2.4.2. Parámetros

Son las variables que se definen en la declaración de un método y sirven para recibir información externa. Actúan como entradas que el método necesita para ejecutar sus instrucciones.[3]

```
public void saludar(String nombre) {
    System.out.println("Hola, " + nombre + "!    Cmo    est s?");
}
```

En el método denominado “saludar” se recibe un parámetro de tipo String llamado “nombre”. Dentro del cuerpo del método, se imprime una cadena de saludo que incluye el valor del parámetro “nombre”. Al llamar a este método, debemos proporcionar un argumento de tipo String que se asignará al parámetro “nombre”.

2.4.3. Argumentos

Son los valores concretos que se pasan a un método cuando este es llamado. Estos valores corresponden a los parámetros definidos en el método. Una vez que hemos declarado un método, podemos llamarlo desde otro lugar de nuestro código. La llamada de un método se realiza utilizando su nombre seguido de paréntesis, y se pueden pasar los argumentos correspondientes si es necesario.[3]

```
int resultado = sumar(5, 3); // Llamada a un metodo con retorno
```

```
public int sumar(int a, int b) {
    return a + b;
}
```

2.5. Encapsulamiento

La encapsulación es un principio fundamental de la programación orientada a objetos (POO) en Java que consiste en agrupar los datos (variables) y los métodos (funciones) que operan sobre los datos en una sola unidad, conocida como clase. Restringe el acceso directo a algunos de los componentes de un objeto y puede evitar la modificación accidental de datos. [4]

La finalidad de la encapsulación es la siguiente:

- Ocultar datos: La encapsulación permite ocultar al exterior la representación interna de un objeto. Sólo se exponen los detalles necesarios a través de una interfaz pública.
- Mayor flexibilidad: Al controlar el acceso a los campos de una clase, puedes cambiar la implementación interna sin afectar al código externo que utiliza la clase.
- Mantenimiento mejorado: La encapsulación ayuda a mantener el código manteniendo los campos privados y proporcionando métodos getter y setter públicos para modificar y ver los campos.

Para lograr la encapsulación en Java:

1. Declara las variables de clase como private.
2. Proporciona los métodos getter y setter de public para acceder y actualizar el valor de una variable privada.

A continuación un ejemplo:

```java
public class Student {
    private String name;
    private int age;

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method for age
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
}
```

En este ejemplo, la clase Student tiene los campos privados name y age. Se proporcionan métodos públicos getter y setter para acceder a estos campos y modificarlos. El configurador de age incluye una comprobación de validación básica.

A continuación un segundo ejemplo con validación:

```java
public class BankAccount {
    private double balance;

    // Getter method for balance
    public double getBalance() {
        return balance;
    }

    // Setter method for balance with validation
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

La clase BankAccount encapsula el campo balance. Proporciona los métodos deposit y withdraw para modificar la balanza, garantizando que sólo se realizan operaciones válidas.

2.6. List y ArrayList

2.6.1. List

Las listas son una de las estructuras de datos básicas en la programación Java y tienen una amplia gama de aplicaciones. Contienen elementos en un orden específico que pueden añadirse, modificarse, borrarse o consultarse. Los objetos de una Java List pueden pertenecer a distintas clases. Además, es posible almacenar elementos duplicados o nulos. Las listas Java admiten clases y métodos genéricos, lo que garantiza la seguridad de tipos. [5]

Las listas se utilizan en aplicaciones de bases de datos para almacenar y acceder a grandes registros de consultas a bases de datos. En las interfaces gráficas de usuario, las listas Java suelen utilizarse para mostrar una lista de elementos, por ejemplo, las opciones de un menú desplegable o los distintos artículos de una tienda online.[5]

Las listas de Java también son indispensables en algoritmos y estructuras de datos. Se utilizan en la implementación de algoritmos de ordenación, algoritmos de búsqueda o estructuras de pilas y colas. En aplicaciones de red, las listas pueden ayudar a facilitar la gestión de conexiones y sockets. Las listas de Java pertenecen a la interfaz collections y deben importarse desde el paquete java.util. Las clases de implementación incluyen Java ArrayList, LinkedList, Vector y Stack. [5]

```
List linkedList = new LinkedList(); // LinkedList
List arrayList = new ArrayList(); // ArrayList
List vecList = new Vector(); // Vector
List stackList = new Stack(); //Stack
```

2.6.2. ArrayList

El ArrayList de Java tiene un tamaño dinámico, lo que significa que los elementos pueden ser fácilmente añadidos o eliminados. Además, la clase ArrayList pertenece al Java Collections Framework y, a diferencia de los arrays, no está disponible de forma nativa. Debe importarse de la biblioteca java.util.[6]

El ArrayList es una opción apropiada cuando la longitud de la lista de Java puede variar. Algunos ejemplos son el almacenamiento de objetos, la búsqueda u ordenación de datos y la creación de listas o colas.[6]

El tamaño del tipo de datos array no puede modificarse. Por lo tanto, el número de objetos que debe contener el array debe conocerse de antemano. Por ello, los arrays son adecuados para gestionar un conjunto predefinido de tipos de datos primitivos como int, float, chat o boolean.[6]

Una desventaja de ArrayList es el mayor tiempo de acceso. Mientras que con los arrays existe una zona de memoria reservada fija, con ArrayList esta no es contigua. Por lo tanto, es importante tener en cuenta las ventajas y desventajas respectivas y seleccionar la estructura de datos adecuada para cada uso. Antes de crear un ArrayList, la clase correspondiente debe ser importada de la librería java.util:[6]

```
import java.util.ArrayList;
```

La sintaxis general es:

```
ArrayList<Type> arrayList= new ArrayList<>();
```

“Type” representa el tipo de datos respectivo de Java.[6]

2.7. Paquetes

Un paquete (package) en Java es un espacio de nombres (namespace) que organiza un conjunto de clases, interfaces, enumeraciones y anotaciones relacionadas. Actúa como una carpeta lógica que agrupa elementos con funcionalidad similar. [7]

Para crear un paquete, debemos utilizar la declaración del paquete agregándola como la primera línea de código en un archivo . [8]

Se recomienda encarecidamente colocar cada nuevo tipo en un paquete. Si definimos tipos y no los colocamos en un paquete, se incluirán en el paquete predeterminado o sin nombre. Usar paquetes predeterminados tiene algunas desventajas: [8]

- Perdemos los beneficios de tener una estructura de paquete y no podemos tener subpaquetes.
- No podemos importar los tipos del paquete predeterminado desde otros paquetes.
- Los ámbitos de acceso protegido y privado del paquete no tendrían sentido.

Por ejemplo, para crear un paquete a partir de www.baeldung.com , invirtámoslo:

```
com.baeldung
```

Comencemos definiendo una clase TodoItem en un subpaquete llamado dominio :

```
package com.baeldung.packages.domain;

public class TodoItem {
    private Long id;
    private String description;

    // standard getters and setters
}
```

Al compilar nuestras clases empaquetadas, debemos recordar la estructura de directorios. Empezando por la carpeta de origen, debemos indicarle a javac dónde encontrar nuestros archivos. [8]

Primero necesitamos compilar nuestra clase TodoItem porque nuestra clase TodoList depende de ella.Comencemos abriendo una línea de comando o terminal y navegando a nuestro directorio de origen. Ahora, compilemos nuestra clase com.baeldung.packages.domain.TodoItem :

```
> javac com/baeldung/packages/domain/TodoItem.java
```

Ahora que nuestra clase TodoItem está compilada, podemos compilar nuestras clases TodoList y TodoApp :

```
>javac -classpath . com/baeldung/packages/*.java
```

Ejecutemos nuestra aplicación usando el nombre completo de nuestra clase TodoApp :

```
>java com.baeldung.packages.TodoApp
```

2.8. Interfaz MainApp

En el desarrollo de aplicaciones orientadas a objetos, un ejemplo representativo es la implementación de un carrito de compras utilizando Java Swing como herramienta de interfaz gráfica. Este tipo de programas sigue el paradigma de la Programación Orientada a Objetos al estructurar las funcionalidades en clases específicas: el modelo, que administra los datos (carrito y artículos); la vista, que corresponde a la interfaz con la cual interactúa el usuario; y el controlador, encargado de enlazar los eventos de la vista con las operaciones del modelo. Este enfoque, cercano al patrón Modelo–Vista–Controlador (MVC), favorece la modularidad y el mantenimiento del código. Asimismo, la interacción se basa en un esquema de eventos y listeners, donde las acciones del usuario (por ejemplo, presionar un botón) generan respuestas predefinidas en el sistema. Para garantizar la robustez, el programa incorpora mecanismos de validación de entradas, evitando excepciones en tiempo de ejecución al comprobar que los datos sean correctos antes de procesarlos. Finalmente, la representación de los resultados incluye el formateo de valores monetarios, lo que asegura una salida clara y estandarizada al usuario. Todo esto refleja cómo la combinación de POO y Swing permite diseñar aplicaciones prácticas que integran lógica de negocio con una interfaz gráfica intuitiva.

3. Desarrollo

3.1. Descripción de la Implementación

Para esta práctica se retomó el código visto en clase, el cual consiste en una pequeña aplicación de un carrito de compras. El objetivo principal fue aplicar el concepto de encapsulamiento en la clase Articulo, además de organizar correctamente el código dentro del paquete mx.unam.fi.poo.p56 para que la aplicación funcionara de acuerdo con las instrucciones dadas.

3.2. Solución teórica: Encapsulamiento

En teoría, el encapsulamiento consiste en proteger los datos de una clase, evitando que puedan ser modificados directamente desde fuera de la misma. Para lograrlo, se declaran los atributos como private y se crean métodos públicos llamados getters y setters, que permiten acceder y modificar los valores de manera controlada. Esto también mejora la mantenibilidad y seguridad del código.

a continuacion la implementacion:

```
1  public class Articulo {
2      String nombre;
3      double precio;
4
5      public Articulo(String nombre, double precio) {
6          this.nombre = nombre;
7          this.precio = precio;
8      }
9
10     public String toItemString() {
11         long cents = Math.round(precio * 100);
12         String entero = String.valueOf(cents/100);
13         int dec = (int)(cents % 100);
14         String decStr = dec < 10 ? ("0" + dec) : String.valueOf(
       dec);
15         return nombre + " - $" + entero + "." + decStr;
16     }
17 }
```

este representa la clase carrito sin haberse modificado con lo cual para concretar el objetivo se realiza lo siguiente:

```java
public class Articulo {
    private String nombre;
    private double precio;

    public Articulo(String nombre, double precio) {
        setNombre(nombre);
        setPrecio(precio);
    }
    public void setNombre(String nombre){
        this.nombre=nombre;

    }
    public void setPrecio(double precio){
        this.precio=precio;
    }
    public String getNombre(){
        return nombre;
    }
    public double getPrecio(){
        return precio;
    }

    public String toItemString() {
        long cents = Math.round(precio * 100);
        String entero = String.valueOf(cents/100);
        int dec = (int)(cents % 100);
        String decStr = dec < 10 ? ("0" + dec) : String.valueOf(
    dec);
        return nombre + " - $" + entero + "." + decStr;
    }
}
```

Como se observa en la primera implementación, la clase Articulo estaba originalmente sin modificaciones: los atributos nombre y precio eran accesibles directamente desde otras clases, y el método toItemString() mostraba el artículo con su precio en formato de texto. Esta versión servía para que la aplicación funcionara correctamente, pero no aplicaba el principio de encapsulamiento, dejando los datos expuestos.

Para concretar el objetivo de la práctica, se realizó la modificación de la clase Articulo tal como se muestra en la segunda implementación. Los atributos se declararon como private y se añadieron los métodos getters y setters correspondientes, permitiendo un acceso controlado a los datos. El método toItemString() se mantuvo para conservar la funcionalidad de mostrar los artículos, adaptándose al uso de los getters cuando es necesario.

Ahora al haber modificado la clase Articulo debemos modifcar la clase Carrito el cual hace uso de la primera, a continuacion se muestra la clase carrito sin modifcar

```java
import java.util.ArrayList;
import java.util.List;

public class Carrito {
    List<Articulo> articulos; // Este atributo tiene que ser privado y final

    public Carrito() {
        this.articulos = new ArrayList<>(); //Polimorismo
    }

    public boolean agregar(Articulo a) {
        if (a == null) return false;
        if (a.nombre.isEmpty()) return false;
        articulos.add(a);
        return true;
    }

    public boolean eliminarPorIndice(int index) {
        if (index < 0) return false;
        if (index >= articulos.size()) return false;
        articulos.remove(index);
        return true;
    }

    public boolean eliminarPorNombre(String nombre) {
        if (nombre == null) return false;
        String target = nombre.trim();
        if (target.isEmpty()) return false;
        for (int i = 0; i < articulos.size(); i++) {
            if (articulos.get(i).nombre.equalsIgnoreCase(target)) {
                articulos.remove(i);
                return true;
            }
        }
        return false;
    }

    public void limpiar() {
        articulos.clear();
    }

    public List<Articulo> getArticulos() {
        return new ArrayList<>(articulos);
    }

    public double getTotal() {
        double total = 0.0;
        for (int i = 0; i < articulos.size(); i++) {
            total += articulos.get(i).precio;
        }
        return total;
    }
}
```

En esta implementación original de la clase Carrito, los métodos acceden directamente a los atributos nombre y precio de los objetos Articulo. Por ejemplo, en agregar() se verifica a.nombre.isEmpty(), y en getTotal() se suma articulos.get(i).precio.

Al modificar la clase Articulo para aplicar encapsulamiento, estos accesos directos dejan de ser válidos, ya que los atributos ahora son privados. Por lo tanto, es necesario adaptar los métodos

de Carrito para que utilicen los getters (getNombre() y getPrecio()) al acceder a los datos de cada artículo. Esta adaptación asegura que Carrito siga funcionando correctamente mientras se mantiene la integridad y seguridad de los datos, cumpliendo con el principio de encapsulamiento.

implementacion:

```java
import java.util.ArrayList;
import java.util.List;

public class Carrito {
    private final List<Articulo> articulos; // Este atributo tiene que ser privado y final

    public Carrito() {
        this.articulos = new ArrayList<>(); //Polimorismo
    }

    public boolean agregar(Articulo a) {
        if (a == null) return false;
        if (a.getNombre().isEmpty()) return false;
        articulos.add(a);
        return true;
    }

    public boolean eliminarPorIndice(int index) {
        if (index < 0) return false;
        if (index >= articulos.size()) return false;
        articulos.remove(index);
        return true;
    }

    public boolean eliminarPorNombre(String nombre) {
        if (nombre == null) return false;
        String target = nombre.trim();
        if (target.isEmpty()) return false;
        for (int i = 0; i < articulos.size(); i++) {
            if (articulos.get(i).getNombre().equalsIgnoreCase(target)) {
                articulos.remove(i);
                return true;
            }
        }
        return false;
    }

    public void limpiar() {
        articulos.clear();
    }

    public List<Articulo> getArticulos() {
        return new ArrayList<>(articulos);
    }

    public double getTotal() {
        double total = 0.0;
        for (int i = 0; i < articulos.size(); i++) {
            total += articulos.get(i).getPrecio();
        }
        return total;
    }
}
```

Como se puede observar en la implementación modificada de la clase Carrito, se realizaron los ajustes necesarios para que funcione correctamente con la clase Articulo ahora encapsulada. Los métodos que antes accedían directamente a los atributos nombre y precio de cada artículo fueron adaptados para utilizar los getters (getNombre() y getPrecio()), lo que permite mantener los datos privados protegidos y controlar su acceso de manera segura.

Además, el atributo que almacena la lista de artículos se declaró como private final, garantizando que la estructura interna del carrito no pueda ser reemplazada desde fuera de la clase, lo que asegura la integridad de los datos. A pesar de estos cambios, la funcionalidad original de la clase se conserva: los métodos para agregar, eliminar, limpiar y calcular el total de los artículos siguen operando correctamente.

3.3. Empaquetado

Para cumplir con la estructura solicitada de la práctica, se organizó la aplicación dentro del paquete mx.unam.fi.poo.p56. Esto se logra simplemente agregando al inicio de cada archivo .java la línea:

```
1  package mx.unam.fi.poo.p56;
```

Con esta instrucción, todas las clases del proyecto se agrupan bajo un mismo espacio de nombres, lo que nos permite tener mayor organización del código y facilita la interacción entre clases dentro del mismo paquete. No se requiere ningún otro cambio adicional; basta con colocar esta línea al principio del archivo, antes de cualquier declaración de clase o importación.

4. Resultados

```
1  package mx.unam.fi.poo.g1.p56;
2
3  public class Articulo {
4      private String nombre;
5      private double precio;
6
7      public Articulo(String nombre, double precio) {
8          setNombre(nombre);
9          setPrecio(precio);
10     }
11
12     public String getNombre() {
13         return nombre;
14     }
15
16     public double getPrecio() {
17         return precio;
18     }
19
20     public void setNombre(String nombre) {
21         this.nombre = nombre;
22     }
23
24     public void setPrecio(double precio) {
25         this.precio = precio;
26     }
27
28     public String toItemString() {
29         long cents = Math.round(precio * 100);
30         String entero = String.valueOf(cents/100);
31         int dec = (int)(cents % 100);
32         String decStr = dec < 10 ? ("0" + dec) : String.valueOf(dec);
33         return nombre + " - $" + entero + "." + decStr;
34     }
35 }
36
```

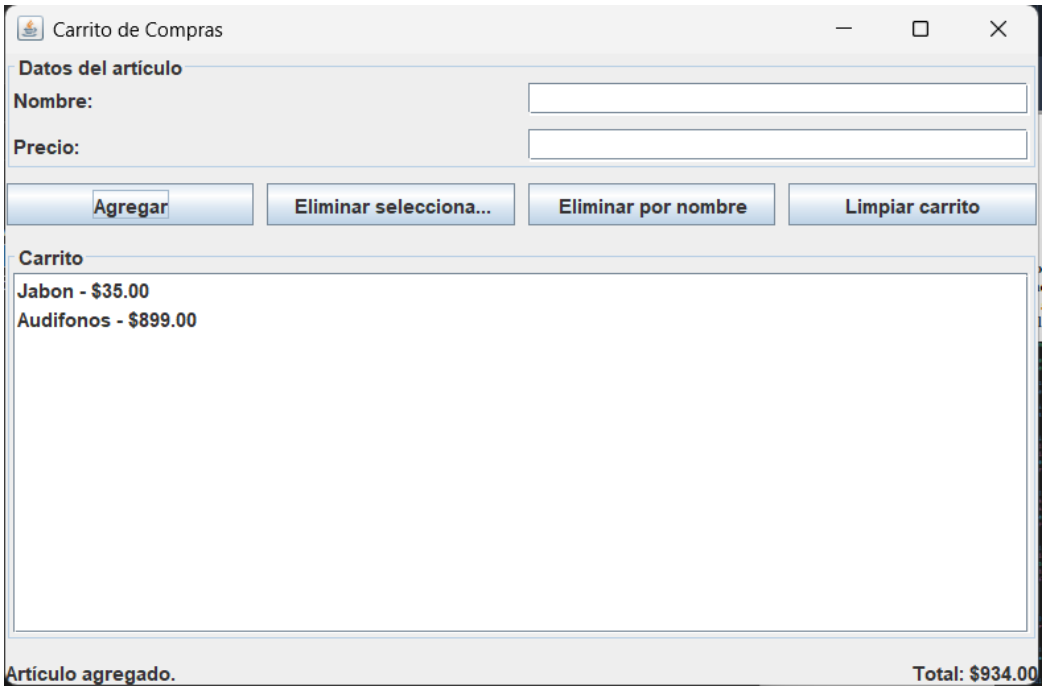
Articulo.java es una clase para representar artículos con nombre y precio, permitiendo obtener su información formateada como cadena, donde se incluyen un constructor para inicializar nombre y precio, getters y setters para acceder y modificar los atributos que están encapsulados y un método *toItemString* que devuelve el artículo como texto en el formato: "nombre - $ entero.decimales"

```java
package mx.unam.fi.poo.g1.p56;

import java.util.ArrayList;
import java.util.List;

public class Carrito {
    private final List<Articulo> articulos;

    public Carrito() {
        this.articulos = new ArrayList<>(); //Polimorismo
    }

    public boolean agregar(Articulo a) {
        if (a == null) return false;
        if (a.getNombre().isEmpty()) return false;
        articulos.add(a);
        return true;
    }

    public boolean eliminarPorIndice(int index) {
        if (index < 0) return false;
        if (index >= articulos.size()) return false;
        articulos.remove(index);
        return true;
    }

    public boolean eliminarPorNombre(String nombre) {
        if (nombre == null) return false;
        String target = nombre.trim();
        if (target.isEmpty()) return false;
        for (int i = 0; i < articulos.size(); i++) {
            if (articulos.get(i).getNombre().equalsIgnoreCase(target)) {
                articulos.remove(i);
                return true;
            }
        }
        return false;
    }

    public void limpiar() {
        articulos.clear();
    }

    public List<Articulo> getArticulos() {
        return new ArrayList<>(articulos);
    }

    public double getTotal() {
        double total = 0.0;
        for (int i = 0; i < articulos.size(); i++) {
            total += articulos.get(i).getPrecio();
        }
        return total;
    }
}
```

Carrito.java es una clase que implementa un carrito de compras básico donde donde se pueden agregar, eliminar, vaciar, consultar los artículos y obtener el total a pagar. Los artículos se guardan en una lista private final que inicia vacía por el constructor y se modificara conforme se utilicen las funciones.

```
PS C:\Users\52551\Programacion\PracticasLab-7> javac -d . mx/unam/fi/poo/g1/p56/*.java
PS C:\Users\52551\Programacion\PracticasLab-7> java mx.unam.fi.poo.g1.p56.MainApp
```

Para ejecutar los códigos, debido a que están empaquetados en mx.unam.fi.poo.g1.p56 se usa el compilador javac con el -d para decir que lo que compilara se realizara dentro de la carpeta actual, compilando todos los archivos .java de esa carpeta y para mandar a llamar la función main solo es necesario añadir el paquete y la clase (MainApp.java).



Al mandar a llamar la función MainApp se desplegara una ventana donde se usan las clases y nos indican las funciones que se asignaron para poder agregar, eliminar, vaciar, consultar los artículos y obtener el total a pagar, pero esta vez visto en una versión con interfaz gráfica.

5. Conclusiones

La practica permitió aplicar de manera integral los conceptos de encapsulamiento y empaquetamiento en el desarrollo del çarrito de compras", modificando las clases de Articulo,java y Carrito.java. Mediante el encapsulamiento se protege la integridad de los atributos, restringiendo su acceso directo y permitiendo la manipulación de los mismos unicamente a través de métodos controlados (getters y setters). Esto no solo garantiza un mayor nivel de seguridad en el manejo de los datos, sino que también facilita la detección y corrección de posibles errores al mantener un control centralizado se las modificaciones.

Por otra parte, el uso de paquetes (mx.unam.fi.poo.g1.p56) representa un paso importante en la organización del código, ya que agrupó de manera lógica las clases relacionadas, mejorando la legibilidad y fomentando la escalabilidad del proyecto. La aplicación del Full Qualified Name fue un complemento clave en este aspecto, al permitir acceder a las clases de manera explícita y sin riesgo de ambigüedad, lo cual resulta especialmente útil cuando se trabaja en proyectos de gran tamaño con múltiples módulos.

En conjunto, estas técnicas demuestran la relevancia de adoptar buenas prácticas de programación, ya que ayudan a generar un código más limpio y reutilizable, ademas preparan al desarrollador para enfrentar proyectos reales en el ámbito profesional. De esta forma, la práctica no se limitó únicamente a cumplir un objetivo académico, sino que también reforzó la importancia de estructurar adecuadamente el software, garantizando que sea mantenible, escalable y alineado con estándares de calidad en la industria.

Referencias

[1] DataCamp. (2025) Clases y objetos en java. Accedido: 16 de septiembre de 2025. [Online]. Available: https://www.datacamp.com/es/doc/java/classes-and-objects

[2] ——. (2025) Java constructors. Accedido: 16 de septiembre de 2025. [Online]. Available: https://www.datacamp.com/es/doc/java/constructors

[3] A. Barragán. (2023) Introducción a java: Métodos, parámetros y argumentos. Publicado: 24 de octubre de 2023; Accedido: 16 de septiembre de 2025. [Online]. Available: https://openwebinars.net/blog/introduccion-a-java-metodos-parametros-y-argumentos/

[4] DataCamp. (2023) Encapsulation in java. Accedido: 30 de septiembre de 2025. [Online]. Available: https://www.datacamp.com/es/doc/java/encapsulation

[5] E. editorial de IONOS. (2023) Java list: métodos y ámbitos de uso más importantes. Accedido: 30 de septiembre de 2025. [Online]. Available: https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/java-list/

[6] I. editorial team. (2023, Mar.) How to use java arraylist. Accedido: 08-septiembre-2025. [Online]. Available: https://www.ionos.mx/digitalguide/paginas-web/desarrollo-web/java-arraylist/

[7] Oracle. (2025) What is a package? Accedido: 30 de septiembre de 2025. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/concepts/package.html

[8] Baeldung. (2024) Guide to java packages. Accedido: 30 de septiembre de 2025. [Online]. Available: https://www.baeldung.com/java-packages